به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

هوش مصنوعی قابل اعتماد

تمرین اول

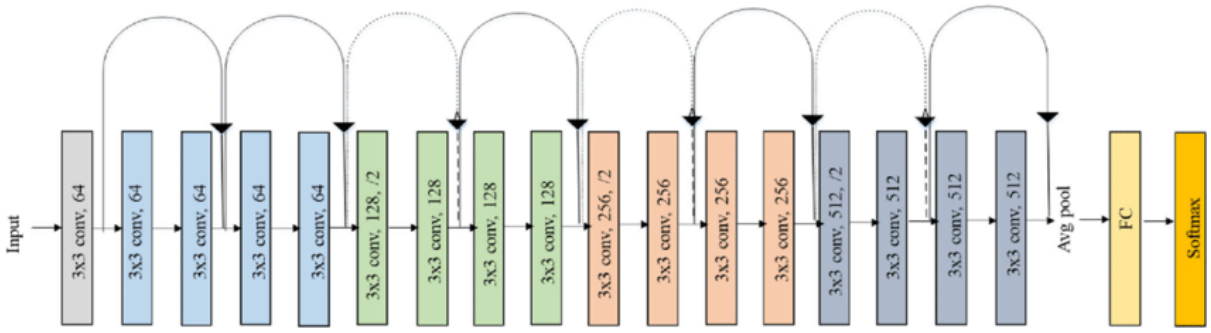
نام و نام خانوادگی	نیما مدیرکیاسرایی
شماره دانشجویی	۸۱۰۱۰۲۳۳۹
تاریخ ارسال گزارش	۱۴۰۳/۰۱/۲۶

فهرست

سؤال اول: Generalization

آموزش مدل اولیه

در این قسمت ابتدا به صورت دستی یک مدل Resnet18 را توسط کتابخانه torch پیاده سازی می کنیم.
این پیاده سازی با کمک معماری این شبکه که در تصویر زیر آمده است انجام می شود.



شکل ۱- معماری شبکه Resnet18

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

شکل ۲- لایه ها و پارامتر های شبکه Resnet18

```
x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.maxpool(x)
x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)
x = self.avgpool(x)
x = torch.flatten(x, 1)
x = self.fc(x)
```

شکل ۳ – بخشی از کد پیاده سازی شبکه Resnet18

در تصویر شماره ۳ می توانیم بخش نهایی کد پیاده سازی شبکه Resnet18 که شامل پشت هم قرار دادن لایه های این شبکه می باشد را مشاهده کنیم.

در ادامه دیتاست SVHN را در دو بخش مختلف train و test لود می کنیم تا بتوانیم از بخش train آن برای آموزش مدل و از بخش test برای ارزیابی مدل استفاده کنیم. (قابل ذکر است که به دلیل امکان استفاده بیشتر از دیتای train و آموزش بهتر مدل، دیتای validation ساخته نشده است.)

```
train_dataset = SVHN(root='svhn_data/', download=True, split = 'train', transform=ToTensor())
test_dataset = SVHN(root='svhn_data/', download=True, split = 'test', transform=ToTensor())
batch_size = 16
train_loader_svhn = DataLoader(train_dataset, batch_size, shuffle=True, num_workers=2)
test_loader_svhn = DataLoader(test_dataset, batch_size, shuffle=False, num_workers=2)
```

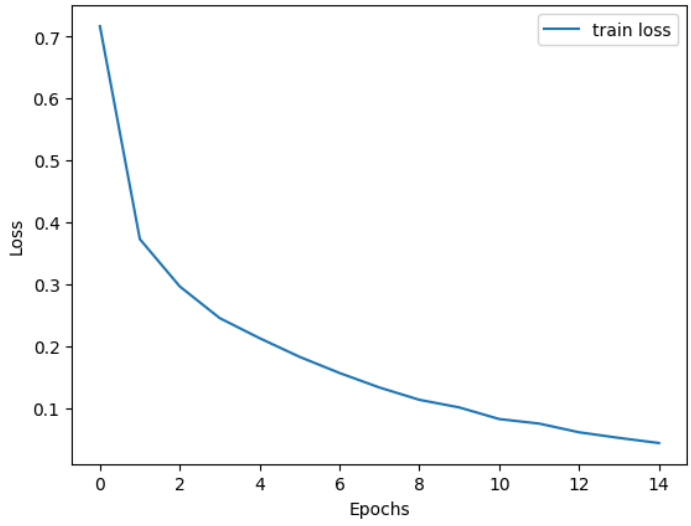
شکل ۴ – لود دیتاست SVHN

در نهایت پارامتر های مورد نظر برای آموزش مدل را به شکل زیر تنظیم می کنیم:

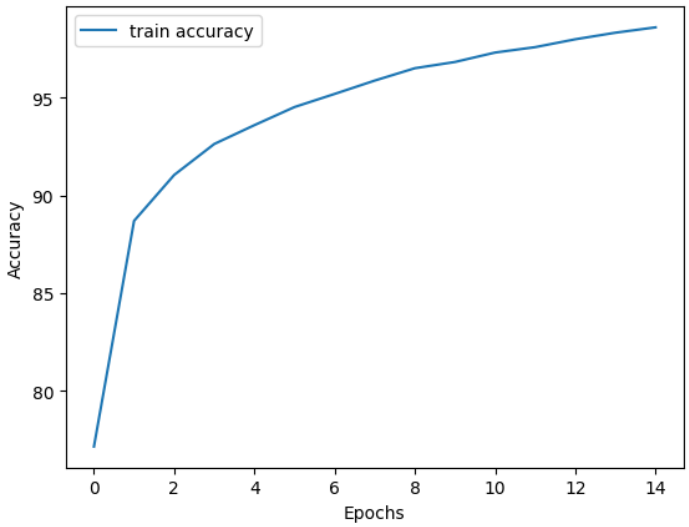
```
epochs = 15
learning_rate = 0.001
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.99)
criterion = nn.CrossEntropyLoss()
```

شکل ۵ – پارامترهای آموزش مدل

در ادامه به آموزش مدل برای ۱۵ ایپاک می پردازیم و نمودار Accuracy و Loss را رسم می کنیم.



شکل ۶ – نمودار خطا



شکل ۷ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training

100%|██████████| 4579/4579 [01:05<00:00, 70.22it/s]
Training loss: 0.044, training acc: 98.598
```

شکل ۸ – ایپاک آخر آموزش مدل

طبق عکس شماره ۸ می توانیم ببینیم که در ایپاک نهایی مقدار خطا به عدد 0.044 و درصد دقت حین آموزش به 98.598% رسیده است.

```
Accuracy of the model on the SVHN test images: 92.57068223724647%
```

شکل ۹ – دقت مدل روی SVHN

در ادامه می خواهیم دقت مدل را بر روی دیتاست MNIST که مدل آن را ندیده است بسنجیم. با توجه به اینکه تصاویر دیتاست MNIST برخلاف SVHN تک کاناله هستند، برای اینکه آن را به مدل بدهیم یا باید مدل را تغییر دهیم تا این نوع تصاویر را بپذیرد یا دیتاست را تغییر دهیم تا ۳ کاناله (RGB) شود که در اینجا راه دوم را انتخاب می کنیم. در واقع از کلاس GrayscaleToRgb داخل transformation هنگام لود دیتاست MNIST استفاده میکنیم تا دیتا آماده وارد شدن به مدل شود.

```python
class GrayscaleToRgb:
    def __call__(self, img):
        return img.convert('RGB')

# Prepare the MNIST Test Data with necessary transformations
transform = Compose([
    GrayscaleToRgb(),
    ToTensor(),
])

mnist_test = MNIST(root='mnist_data/', train=False, download=True, transform=transform)
test_loader_mnist = DataLoader(mnist_test, batch_size=16, shuffle=False)
```

شکل ۱۰ – لود کردن دیتاست MNIST

در نهایت دقت مدل را بر روی این دیتاست می سنجیم.

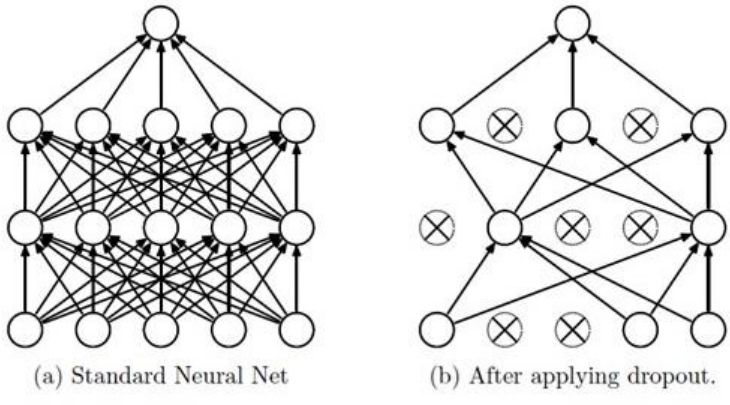
```
Accuracy of the model on the MNIST test images: 55.38%
```

شکل ۱۱ – دقت مدل روی MNIST

بهبود تعمیم پذیری مدل – Model Architecture

در این قسمت ابتدا به بررسی استفاده از dropout و batch normalization می پردازیم و سپس تاثیر حذف لایه batch normalization در مدل Resnet18 را مشاهده می کنیم.

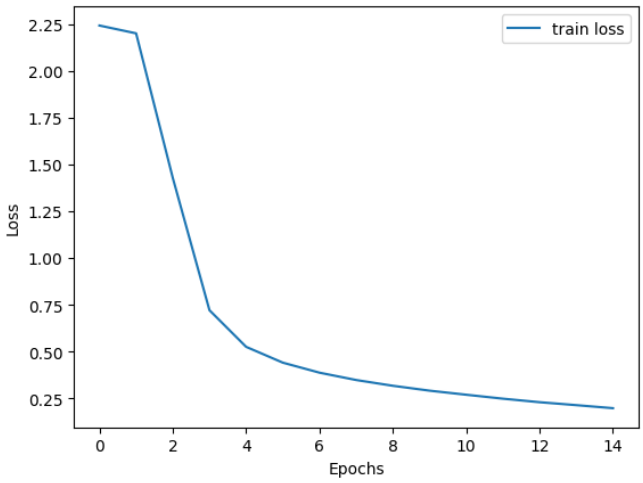
۱. **Dropout**: شبکه‌های عصبی عمیق برای کار کردن به تعداد زیادی پارامتر نیاز دارند که این ویژگی در عین کارآمد شدن باعث ایجاد بیش‌برازش (Overfitting) در آنها می‌شود. برای مقابله با این مشکل، روش Dropout یا حذف کردن معرفی شده است که در حین آموزش واحدهایی از شبکه را به صورت تصادفی حذف می‌کند تا از آموزش زیاد مدل جلوگیری شود. شبکه‌های عصبی با اعمال Dropout می‌توانند با استفاده از کاهش گرادیان تصادفی به شیوه‌ای مشابه شبکه‌های عصبی استاندارد آموزش داده شوند. تنها تفاوت این است که برای هر مورد آموزشی در یک دسته کوچک، ما برخی از واحدها را حذف (Dropout) می‌کنیم تا یک شبکه با ساختار نازک‌تر ایجاد شود. گرادیان‌ها برای هر پارامتر بر روی موارد آموزش در هر دسته کوچک متوسط می‌شوند. هر مورد آموزشی که از یک پارامتر استفاده نمی‌کند، گرادیان صفری برای آن پارامتر ارائه می‌دهد. برای استفاده از Dropout در شبکه‌های پیش‌آموزی‌شده، روش پیش‌آموزی بدون تغییر باقی می‌ماند. وزن‌های به دست آمده از پیش‌آموزی باید با ضریبی افزایش یابند. این کار باعث می‌شود که خروجی مورد انتظار از هر واحد در حالت تصادفی Dropout با خروجی در طول پیش‌آموزی هم‌تراز باشد. ممکن است نرخ یادگیری در مرحله تنظیم مجدد به نرخ‌های یادگیری بهتری برای شبکه‌های از ابتدا تصادفی نزدیک باشد، اما با انتخاب نرخ‌های یادگیری کوچک‌تر، اطلاعات موجود در وزن‌های پیش‌آموزی حفظ می‌شود و امکان بهبود در دقت تعمیم نهایی را بهبود می‌بخشد.



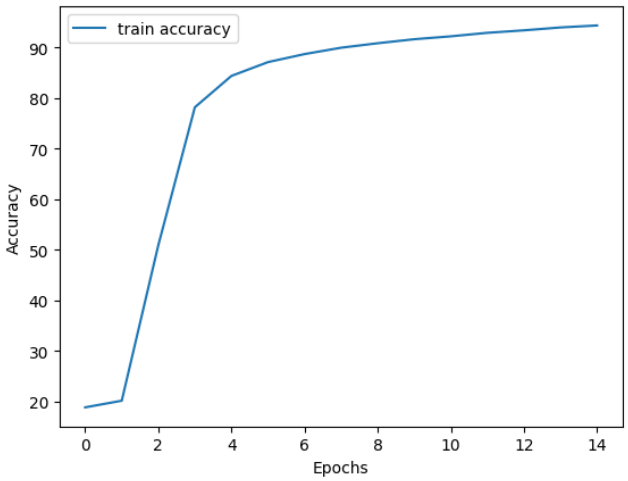
شکل ۱۲ – عملکرد Dropout

Batch normalization: فرایند اموزش بخاطر این مساله که ورودی هر لایه توسط پارامترهای تمامی لایه های قبلی تحت تاثیر قرار میگیره پیچیده میشه. چرا که تغییرات کوچک در پارامترهای شبکه با عمیق تر شدن شبکه تاثیرشون چندین برابر میشه. این تغییرات در توزیع ورودی لایه ها یه مشکلی رو ایجاد میکنه چون لایه ها بطور دائم باید خودشون رو با توزیع جدید تطبیق بدن. زمانی که توزیع ورودی به یک سیستم یادگیری تغییر کنه ما اصطلاحا میگیم covariate shift اتفاق افتاده . در نتیجه ویژگی های توزیع ورودی که موجب بهینه سازی فرآیند آموزش میشه همین تاثیر رو بر روی زیرشبکه هم خواهد داشت. به همین صورت ثابت موندن توزیع x در طی زمان مفید فایده خواهد بود. چرا که با این شرط دیگه پارامترهایی مثل W که در لایه های عمیقتر شبکه هستن لازم نیستد که خودشون رو دائما با تغییرات ورودی وفق بدن . حذف این مشکل یا کاهش اون باعث سرعت در اموزش میشه . روش BN با ترمیم میانگین و واریانس ورودی لایه ها (نرمال کردن توزیع دیتا به شکلی که میانگینش حتما صفر و واریانسش حتما ۱ باشه) به این مهم دست پیدا میکنه . بچ نرمالیزیشن دارای یک اثر مفید بر روی جریان گرادیان در داخل شبکه است که این کار رو از طریق کاهش وابستگی گردایان ها به اسکیل پارامترها و یا مقادیر اولیه اونها انجام میده . این نکته باعث میشه بشه از نرخ یادگیری بیشتری استفاده کرد و سرعت آموزش شبکه رو بیشتر کرد. در زمان آموزش با بچ نرمالیزیشن یک نمونه به همراه سایر نمونه ها در داخل یک مینی بچ بهش نگاه میشه و دیگه اموزش شبکه یک جواب مشخص برای یک مثال اموزشی ایجاد نمیکنه . در نتیجه این حالت باعث تعمیم بهتر شبکه میشه (Generalization) .

۲. در این بخش ابتدا تمام لایه های batch normalization در مدل خود را حذف می کنیم و سپس مجدد شبکه را آموزش می دهیم.



شکل ۱۳ – نمودار خطا



شکل ۱۴ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training

100%|██████████| 4579/4579 [00:50<00:00, 90.40it/s]
Training loss: 0.197, training acc: 94.310
```

شکل ۱۵ – اپاک آخر آموزش مدل

طبق عکس بالا می توانیم ببینم که دقت نهایی حین آموزش نسبت به حالتی که BN در شبکه وجود داشت، ۴ درصد پایین تر آمده و مقدار خطای آموزش هم افزایش پیدا کرده.

همچنین اگر به نمودارهای دقت و خطا نگاه کنیم می توانیم ببینیم که دقت از تقریبا ۱۸٪ شروع کرده و به بالای ۹۰ درصد رسیده و عدد خطا از تقریبا ۲.۲۵ به زیر ۰.۲ رسیده است. در صورتی که برای حالت قبل (وجود BN در شبکه) تغییرات دقت و خطا در بازه کوچکتری رخ می داد به عنوان مثال دقت از بالای ۸۰ درصد شروع می شد.

```
Accuracy of the model on the SVHN test images: 91.9099569760295%
```

شکل ۱۶ – دقت مدل روی SVHN

```
Accuracy of the model on the MNIST test images: 49.34%
```

شکل ۱۷ – دقت مدل روی MNIST

در نهایت می توانیم مشاهده کنیم که با برداشتن BN، دقت مدل بر روی هر دو دیتاست به نسبت حالت قبل کاهش یافته است و مطابق انتظار ما پیش رفته است.

بهبود تعمیم پذیری مدل – Loss Function

روش Label Smoothing یک تکنیک در یادگیری عمیق است که برای کاهش اعتماد بیش از حد مدل به پیش‌بینی‌های خود استفاده می‌شود. این روش با تعدیل برچسب‌های هدف به گونه‌ای که ترکیبی از برچسب درست و توزیع یکنواخت بر روی تمام برچسب‌ها باشد، انجام می‌پذیرد. در نتیجه، مدل کمتر مستعد یادگیری نویز یا خطاهای داده‌ها می‌شود و به این ترتیب عمومیت (Generalization) بیشتری پیدا می‌کند.

وقتی این الگوریتم را با تابع خطای CrossEntropy ترکیب می‌کنیم، عملکرد این تابع به این صورت تغییر می‌کند که به جای استفاده از برچسب‌های صریح و قطعی، از برچسب‌هایی با احتمالات تعدیل شده استفاده می‌شود. به عبارت دیگر، برای هر کلاس به جای اینکه احتمال ۱ به کلاس صحیح و ۰ به سایر کلاس‌ها داده شود، مقداری از احتمال به سایر کلاس‌ها اختصاص داده می‌شود و از احتمال کلاس صحیح کمی کاسته می‌شود. این کار از بیش‌برازش جلوگیری کرده و مدل را مجبور می‌کند که نسبت به داده‌هایی که ممکن است کمی متفاوت باشند یا نویز داشته باشند، مقاوم‌تر شود.

$$y\_ls = (1 - \alpha) * y\_hot + \alpha / K$$

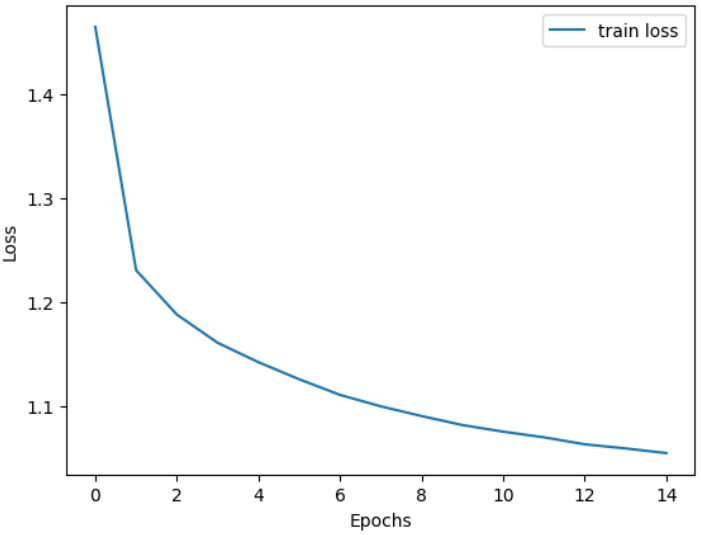
شکل ۱۸ – رابطه Label Smoothing

در ادامه با استفاده از کتابخانه torch تابع خطای CrossEntropy همراه با تکنیک Label Smoothing را پیاده سازی می کنیم و مجدد مدل را با این تابع خطا آموزش می دهیم.

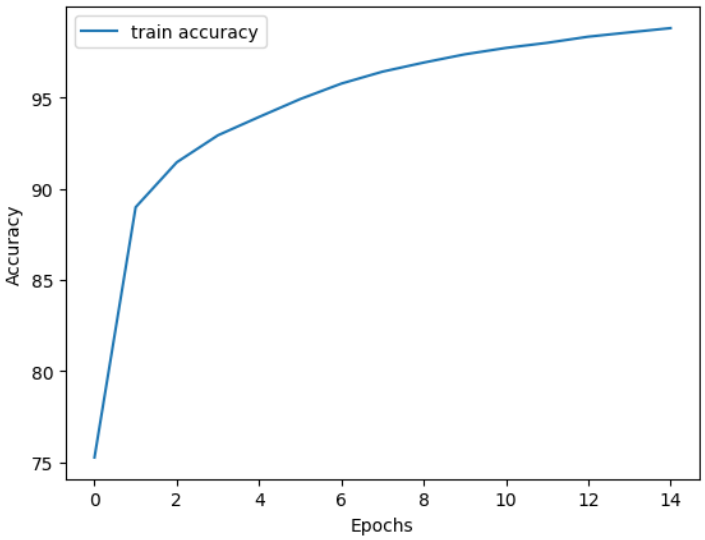
```python
class LabelSmoothingCrossEntropy(nn.Module):
    def __init__(self, smoothing):
        super(LabelSmoothingCrossEntropy, self).__init__()
        self.smoothing = smoothing

    def forward(self, input, target):
        log_probs = F.log_softmax(input, dim=-1)
        n_classes = input.size(-1)
        true_dist = torch.zeros_like(log_probs)
        true_dist.fill_(self.smoothing / (n_classes - 1))
        true_dist.scatter_(1, target.data.unsqueeze(1), 1.0 - self.smoothing)
        return torch.mean(torch.sum(-true_dist * log_probs, dim=-1))
```

شکل ۱۹ – کد پیاده سازی Label Smoothing برای CrossEntropy Loss Function



شکل ۲۰ – نمودار خطا



شکل ۲۱ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training

100%|██████████| 4579/4579 [01:05<00:00, 69.86it/s]
Training loss: 1.055, training acc: 98.801
```

شکل ۲۲ – اپیاک آخر آموزش مدل

همانطور که در شکل بالا می بینیم دقت مدل روی داده ی آموزش نسبت به حالت اول به مقدار خیلی اندکی افزایش داشته است. همچنین مقدار خطا در اپیاک آخر بر روی داده ی آموزش نیز مقدار خوبی زیاد شده است و مطابق انتظار است.

```
Accuracy of the model on the SVHN test images: 92.95482483097726%
```

شکل ۲۳ – دقت مدل روی SVHN

```
Accuracy of the model on the MNIST test images: 61.06%
```

شکل ۲۴ – دقت مدل روی MNIST

همانطور که در شکل بالا می بینیم دقت مدل همراه با روش Label Smoothing بر روی داده ی تست هر دو دیتاست افزایش یافته است. قابل ذکر است که با استفاده از این روش انتظار Generalization بهتر از مدل داشتیم که می بینیم دقت مدل بر روی دیتاست MNIST افزایش یافته است و انتظار ما را برآورده کرده است.

بهبود تعمیم پذیری مدل – Data Augmentation

در این قسمت می خواهیم تاثیر افزایش دیتای آموزش (Data Augmentation) را بر روی دقت و عمومیت مدل بسنجیم. با توجه به اینکه در Augmentation تصاویر، باید لیبل یا کلاس مرتبط با آن ها ثابت بماند و چون دیتاستی که ما با آن کار می کنیم مربوط به تصاویر اعداد می باشد، بنابراین بعضی از انواع افزایش دیتا منجر به تغییر عدد به عدد دیگری می شود که کار اشتباهی است. به عنوان مثال چرخش عمودی (Vertical Flip) باعث جابجا شدن اعداد 6 و 9 می شود. یا چرخش افقی ممکن است باعث جابجایی اعداد 2 و 5 شود زیرا ممکن است شبیه یکدیگر شوند و مدل را به اشتباه بیندازند.

در نهایت با آزمون و خطا ترکیبی از بهترین Augmentation ها را پیاده سازی کرده ایم که در ادامه به آن ها اشاره می کنیم.

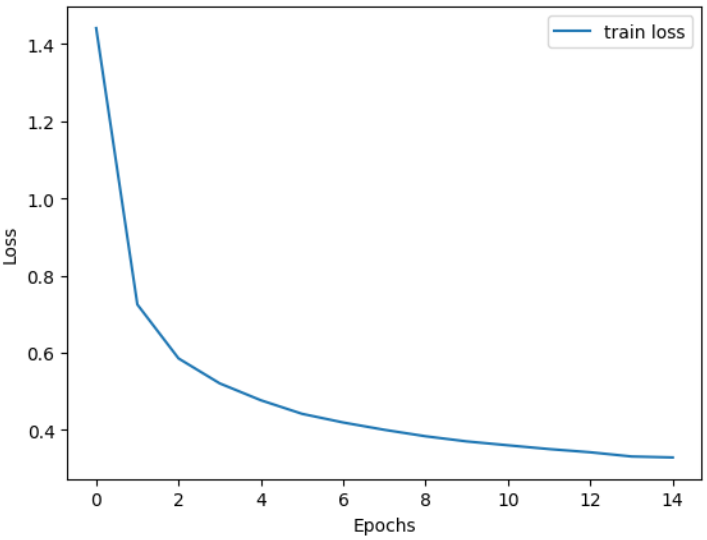
```
transform = transforms.Compose([
    transforms.RandomAffine(degrees=10, translate=(0.1, 0.1), scale=(0.9, 1.1)),  # Rotation, Translation, Scaling
    transforms.ColorJitter(brightness=0.2, contrast=0.2),  # Brightness and Contrast Adjustments
    transforms.RandomPerspective(distortion_scale=0.2, p=0.5),  # Perspective Transformation
    transforms.RandomCrop(size=(32, 32), padding=4),  # Cropping
    transforms.ToTensor(),  # Convert PIL Image to PyTorch Tensor
])
```

شکل ۲۵ – کد پیاده سازی Data Augmentation

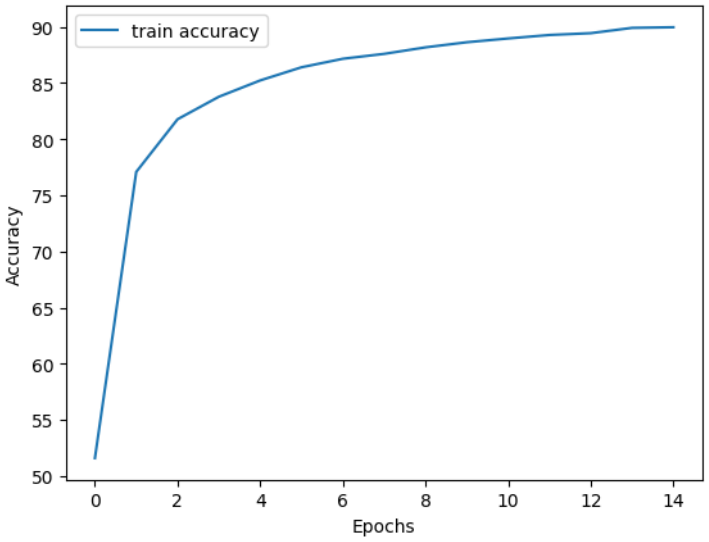
تکنیک های مورد استفاده در Augmentation به ترتیب عبارتند از:

۱. چرخش ۱۰ درجه (که منجر به اشتباه بین اعداد نیز نشود) – جابجایی – اسکیل تصویر
۲. تغییر روشنایی و کنتراست تصاویر
۳. تغییر Perspective تصاویر
۴. برش تصاویر

در ادامه مدل را همراه به این تکنیک ها مجدد آموزش می دهیم و نتایج را مشاهده می کنیم.



شکل ۲۶ – نمودار خطا



شکل ۲۷ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training
100%|██████████| 4579/4579 [01:58<00:00, 38.57it/s]
Training loss: 0.328, training acc: 89.970
```

شکل ۲۸ – ایپاک آخر آموزش مدل

طبق تصویر شماره ۲۸ می توانیم ببینیم که دقت مدل روی تصاویر آموزش چند درصد نسبت به حالت اول پایین تر آمده و دلیل آن هم دیدن نمونه های مختلف و افزایش یافته از تصاویر دیتاست می باشد.

```
Accuracy of the model on the SVHN test images: 94.08036263060848%
```

شکل ۲۹ – دقت مدل روی SVHN

```
Accuracy of the model on the MNIST test images: 57.11%
```

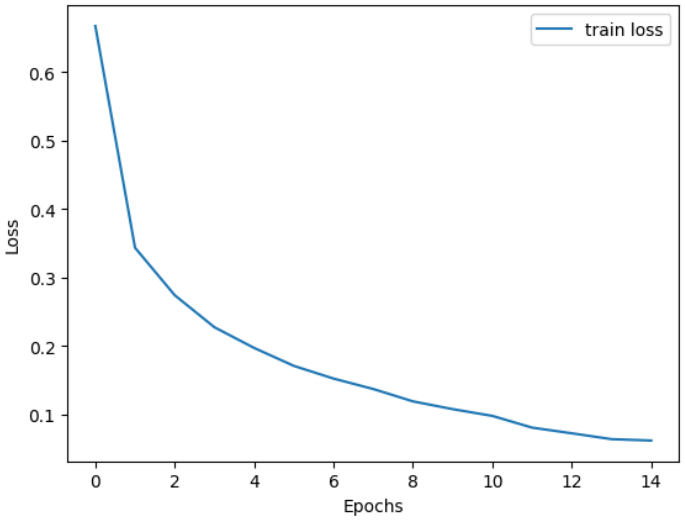
شکل ۳۰ – دقت مدل روی MNIST

همانطور که در تصاویر شماره ۲۹ و ۳۰ می بینیم دقت مدل روی هر دو دیتاست مورد نظر نسبت به حالت اولیه (بدون Augmentation) افزایش داشته است و توانسته ایم عمومیت مدل را با این تکنیک افزایش دهیم.

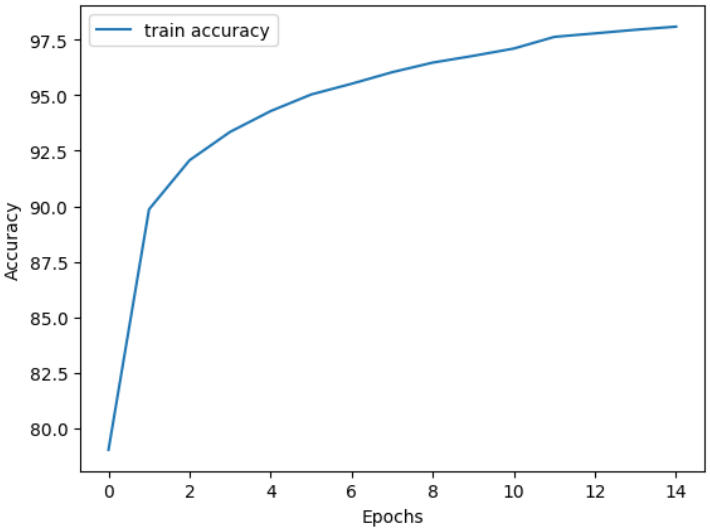
نکته دیگری که قابل توجه می باشد افزایش زمان تقریبا یک دقیقه ای ران شدن هر ایپاک نسبت به حالت اولیه است(بدون Augmentation) که به دلیل تبدیلات مختلف منطقی است.

بهبود تعمیم پذیری مدل – Input Features (Feature Extraction)

در این قسمت ابتدا مدل از پیش آموزش دیده شده Resnet18 را لود می کنیم و سپس مجدد مدل را آموزش می دهیم تا تاثیر وزن های ImageNet را مشاهده کنیم.



شکل ۳۱ – نمودار خطا



شکل ۳۲ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training
100%|██████████| 4579/4579 [00:57<00:00, 79.90it/s]
Training loss: 0.062, training acc: 98.089
```

شکل ۳۳ – ایپاک آخر آموزش مدل

طبق عکس شماره ۳۳ می توان دید که دقت و خطا به نسبت حالت اولیه (مدل خام) تغییر چندانی نداشته است اما در ایپاک اول از دقت بالاتری شروع شده است که دلیل آن وجود وزن های اولیه ImageNet می باشد.

```
Accuracy of the model on the SVHN test images: 92.70897357098956%
```

شکل ۳۴ – دقت مدل روی SVHN

```
Accuracy of the model on the MNIST test images: 61.3%
```

شکل ۳۵ – دقت مدل روی MNIST

طبق تصاویر ۳۴ و ۳۵ می توانیم ببینیم که دقت مدل همراه با وزن های ImageNet به دلیل استخراج ویژگی های غنی بهتر از مدل خام می باشد. همچنین می توانیم ببینیم که دقت آن بر روی دیتاست دیده نشده نیز افزایش یافته است که نشان از بهبود Generalization مدل می دهد.

بهبود تعمیم پذیری مدل – Adam Optimizer

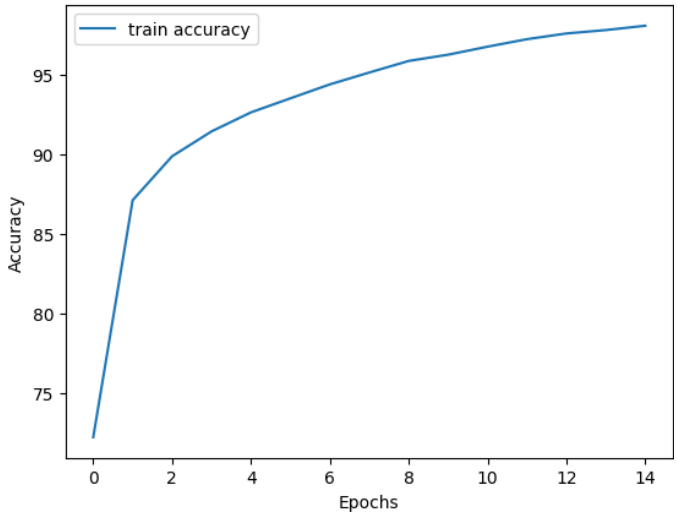
در این قسمت به جای استفاده از بهینه ساز SGD از Adam استفاده می کنیم. به طور کلی نمی توان گفت کدام بهینه ساز بهتری است اما ممکن است در مسائل و دیتاهای مختلف هر کدام از آنها بهتر از دیگری نتیجه دهد. به طور کلی برخی مزایای Adam نسبت به SGD عبارتند از:

تنظیم خودکار نرخ یادگیری: Adam نرخ یادگیری را برای هر پارامتر به طور جداگانه تنظیم می‌کند بر اساس میانگین متحرک از دومین مشتق تابع هدف (مربعات گرادیان‌ها). این ویژگی به آن اجازه می‌دهد که در طول فرآیند آموزش نرخ یادگیری را بهینه سازد، در حالی که SGD با یک نرخ یادگیری ثابت کار می‌کند.

مقاومت در برابر نویز داده‌ها و ناهمواری‌های تابع هدف: Adam می‌تواند بهتر با داده‌های نویزدار و توابع هدف پیچیده که دارای ناهمواری‌های زیاد هستند کنار بیاید. این امر به خصوص در دیتاست‌های بزرگ و پیچیده مانند CIFAR10 مفید است.

سرعت همگرایی سریع‌تر: Adam معمولاً سریع‌تر از SGD به همگرایی می‌رسد به خاطر استفاده از مکانیزم‌هایی مانند ممان اول و دوم که به آن اجازه می‌دهد تا تنظیمات بهتری را برای گام‌های یادگیری انجام دهد.

کاهش احتمال گیر افتادن در مینیمم های محلی: به دلیل تنظیم خودکار نرخ یادگیری و استفاده از ممان‌ها، Adam ممکن است کمتر در مینیمم های محلی گیر کند، که این موضوع می‌تواند در مسائل پیچیده‌تر کاربردی‌تر باشد.



شکل ۳۶ – نمودار خطا

شکل ۳۷ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training
100%|██████████| 4579/4579 [01:03<00:00, 72.13it/s]
Training loss: 0.062, training acc: 98.078
```

شکل ۳۸ – ایپاک آخر آموزش مدل

```
Accuracy of the model on the SVHN test images: 92.44775660725261%
```

شکل ۳۹ – دقت مدل روی SVHN

```
Accuracy of the model on the MNIST test images: 64.96%
```

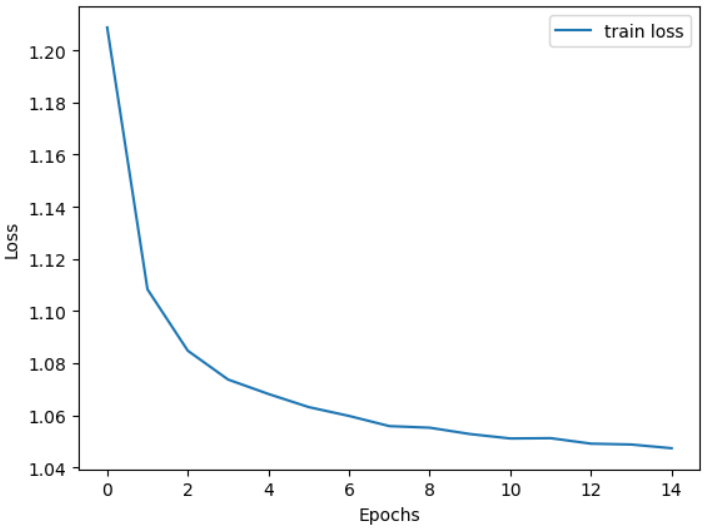
شکل ۴۰ – دقت مدل روی MNIST

طبق تصاویر بالا می توانیم ببینیم که دقت مدل روی دیتاست آموزش داده شده (SVHN) تغییر آنچنانی نکرده است اما دقت آن بر روی دیتاست دیده نشده تقریبا ۶ درصد افزایش یافته است و مدل عمومی تر شده است.

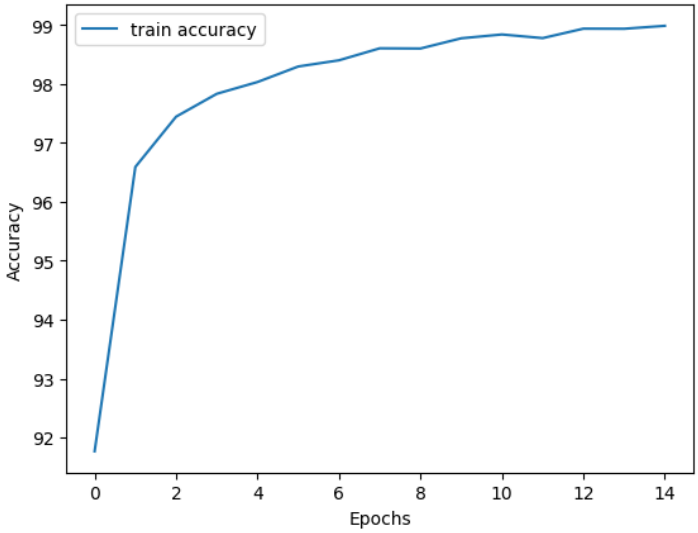
آموزش معکوس مدل – Unsupervised

در اینجا مدل را با بهترین setting در قسمت های قبل تعریف می کنیم و روی دیتاست MNIST آموزش می دهیم. بهترین setting دست یافته در قسمت های قبل عبارت است از:

شبکه همراه با Batch Normalization – تابع خطا همراه با Label Smoothing – تکنیک های Augmentation گفته شده در قسمت های قبل – لود کردن مدل همراه با وزن های ImageNet – استفاده از بهینه ساز Adam



شکل ۴۱ – نمودار خطا



شکل ۴۲ – نمودار دقت

```
[INFO]: Epoch 15 of 15
Training
100%|██████████| 3750/3750 [01:41<00:00, 37.11it/s]
Training loss: 1.047, training acc: 98.987
```

شکل ۴۳ – اپیاک آخر آموزش مدل

```
Accuracy of the model on the MNIST test images: 99.16%
```

شکل ۴۴ – دقت مدل روی MNIST

```
Accuracy of the model on the SVHN test images: 23.14843269821758%
```

شکل ۴۵ – دقت مدل روی SVHN

همانطور که در تصاویر بالا می بینیم دقت مدل بر روی دیتاست MNIST خیلی بالا (تقریبا ۹۹٪) است و به خوبی ویژگی های این دیتاست را یاد گرفته است اما دقت آن بر روی داده های تست دیتاست SVHN، ۲۳٪ است که اصلا عدد خوب و قابل قبولی نیست و می توان گفت که دقت اصلا شبیه به حالت های قبل نیست. در ادامه به چند مورد از علت های این اتفاق اشاره می کنیم:

پیچیدگی دیتاست‌ها: SVHN دیتاستی پیچیده‌تر و متنوع‌تر نسبت به MNIST است. SVHN شامل تصاویر رنگی از اعداد خانه‌ها است که از Google Street View گرفته شده‌اند. تصاویر در زمینه‌های پس‌زمینه، موقعیت‌ها، نورپردازی متنوع هستند که این دیتاست را بسیار نزدیک‌تر به سناریوهای واقعی جهان نسبت به MNIST می‌کند. از طرف دیگر، MNIST شامل تصاویر سیاه و سفید از اعداد دست‌نویس با اندازه یکسان و پس‌زمینه‌های نسبتاً ساده است. این سادگی به این معناست که ویژگی‌های یادگرفته شده از MNIST ممکن است غنی یا متنوع نباشند تا بتوانند با پیچیدگی SVHN کنار بیایند.

توانایی عمومی‌سازی: هنگامی که روی SVHN آموزش داده می‌شود، مدل ویژگی‌های پیچیده‌تر، متنوع‌تر و مقاوم‌تری را یاد می‌گیرد که احتمالاً قادر به عمومی‌سازی بهتر به دیگر دیتاست‌ها، از جمله MNIST، با وجود تغییر در توزیع داده‌ها می‌باشد. مدل ResNet18 آموزش‌دیده روی SVHN یاد گرفته است تا انواع مختلفی از سبک‌های عددی را در شرایط مختلف شناسایی کند، که هنوز هم تا حدی هنگام مواجهه با اعداد ساده‌تر MNIST کاربرد دارد.

انتقال‌پذیری ویژگی‌ها: ویژگی‌های یادگرفته شده از MNIST بسیار ابتدایی و مخصوص به سبک اعداد دست‌نویس دیده شده در آن دیتاست هستند. این ویژگی‌ها قابل انتقال به SVHN نیستند که تصاویری با پیچیدگی بصری بیشتر دارد. وقتی مدلی که روی MNIST آموزش دیده روی SVHN آزمایش می‌شود، به دلیل ناکافی بودن ویژگی‌های یادگرفته شده (تشخیص ساده خطوط، اشکال ابتدایی و غیره) در مقابله با پیچیدگی‌های تصاویر طبیعی دچار مشکل می‌شود.

اندازه و تنوع دیتاست: MNIST کوچکتر و کم‌تنوع‌تر از SVHN است. آموزش روی دیتاستی بزرگ‌تر و متنوع‌تر معمولاً به ساخت مدلی کمک می‌کند که عملکرد بهتری روی داده‌های دیده نشده داشته باشد.

آموزش معکوس مدل – Supervised

در این قسمت، همان کار قسمت قبل را انجام می دهیم صرفا قبل از تست کردن مدل، به اندازه ۱۰ ایپاک classifier نهایی مدل را روی ۸۰۰ داده رندوم دیتاست SVHN آموزش می دهیم تا تاثیر آن را مشاهده کنیم.

```
# Freeze all layers first
for param in model.parameters():
    param.requires_grad = False

# Unfreeze the fully connected layers
for param in model.fc.parameters():
    param.requires_grad = True
```

شکل ۴۶ – freeze کردن لایه های کانولوشنی برای عملیات fine tune

قبل از fine tune کردن مدل، مطابق شکل شماره ۴۶ ابتدا کل پارامترها و لایه های مدل را freeze می کنیم (غیر قابل آموزش) و سپس فقط پارامترهای لایه آخر مدل (Fully Connected) را از freeze در می آوریم تا بتواند آموزش ببیند و تغییر کند.

```
Accuracy of the model on the MNIST test images: 74.58%
```

شکل ۴۷ – دقت مدل روی MNIST

```
Accuracy of the model on the SVHN test images: 28.196066379840197%
```

شکل ۴۸ – دقت مدل روی SVHN

طبق تصاویر ۴۷ و ۴۸ می توان گفت که fine tune کردن لایه آخر مدل باعث کاهش دقت روی تصاویر MNIST و افزایش چند درصدی دقت روی تصاویر SVHN شده است که مطابق چیزی است که انتظار داشته ایم. در واقع مدل توانسته به مقدار اندکی بهتر ویژگی های دیتای SVHN را استخراج کند. همچنین احتمالا اگر مدل را با تعداد ایپاک بیشتر و یا داده بیشتری fine tune کنیم می توانیم به دقت بیشتری بر روی دیتاست SVHN برسیم.

سؤال دوم: Robustness

در این قسمت قصد داریم تا با حملات متخاصمانه بیشتر آشنا شویم و مقاومت دادگان CIFAR10 را نسبت به این نوع حملات بررسی و افزایش دهیم.

در ابتدا دیتاست CIFAR10 را طبق همان درصدهای گفته شده در صورت سوال و مدل از قبل آموزش دیده شده Resnet18 را لود می کنیم.

```
transform = transforms.Compose([
    transforms.Resize(224),  # Resize images to fit ResNet18's expected input size
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load the CIFAR10 dataset
dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
val_size = 10000
train_size = len(dataset) - val_size
train_dataset, valid_dataset = random_split(dataset, [train_size, val_size])

batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2)

device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = models.resnet18(pretrained=True)
num_classes = 10
model.fc = torch.nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)
```

شکل ۴۹ – لود دیتاست CIFAR10 و مدل Resnet18

لازم به ذکر است که در خطوط آخر کد عکس شماره ۴۹ ابتدا مدل مورد نظر لود شده است و سپس تعداد ویژگی های خروجی لایه آخر (Fully Connected) به تعداد ۱۰ (تعداد کلاس های دیتاست CIFAR10) تغییر داده شده است زیر به صورت دیفالت این عدد برای مدل Resnet18 برابر با ۱۰۰۰ می باشد.

در ادامه به توضیح حملات FGSM و PGD می پردازیم.

حملات Fast Gradient Sign Method (FGSM):

FGSM برای حمله به شبکه‌های عصبی با نفوذ به روش یادگیری آن‌ها یعنی «گرادیان» (Gradients) طراحی شده است. این الگوریتم به جای کار برای کمینه کردن «زیان» (Loss) با تنظیم وزن‌ها بر پایه «گرادیان بازگشت به عقب» (backpropagated gradients)، حمله داده‌های ورودی را با هدف بیشینه‌سازی زیان برپایه بازگشت به عقب (Backpropagation) مشابهی تنظیم می‌کند. به عبارت دیگر، حمله از گرادیان زیان داده ورودی استفاده و سپس داده ورودی را برای بیشینه کردن زیان تنظیم می‌کند. فرمول این روش به صورت زیر است:

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

در رابطه بالا x نشان دهنده ورودی اصلی، $\acute{x}$ ورودی مخرب، ϵ ضریب حمله (مقدار کوچک)، $\nabla_x J(\theta, x, y)$ گرادیان تابع زیان J نسبت به ورودی x، θ پارامترهای مدل و y برچسب صحیح ورودی می باشد.

حملات Projected Gradient Descent (PGD):

PGD یک روش تکرار شونده است که نسخه قوی‌تری از FGSM محسوب می‌شود. این روش با اعمال چندین گام کوچک تغییر و اجبار تغییرات برای باقی ماندن در یک محدوده معین، سعی در فریب دادن شبکه دارد. اما یکی از بدی هایی که نسبت به روش FGSM دارد، هزینه محاسباتی بالای آن به دلیل روش تکرار شونده بودن آن و محاسبه گرادیان به صورت تکرار شونده می باشد. فرمول این روش به صورت زیر است:

$$x'_{t+1} = \text{Proj}_{x+S}(x'_t + \alpha \cdot \text{sign}(\nabla_x J(\theta, x'_t, y)))$$

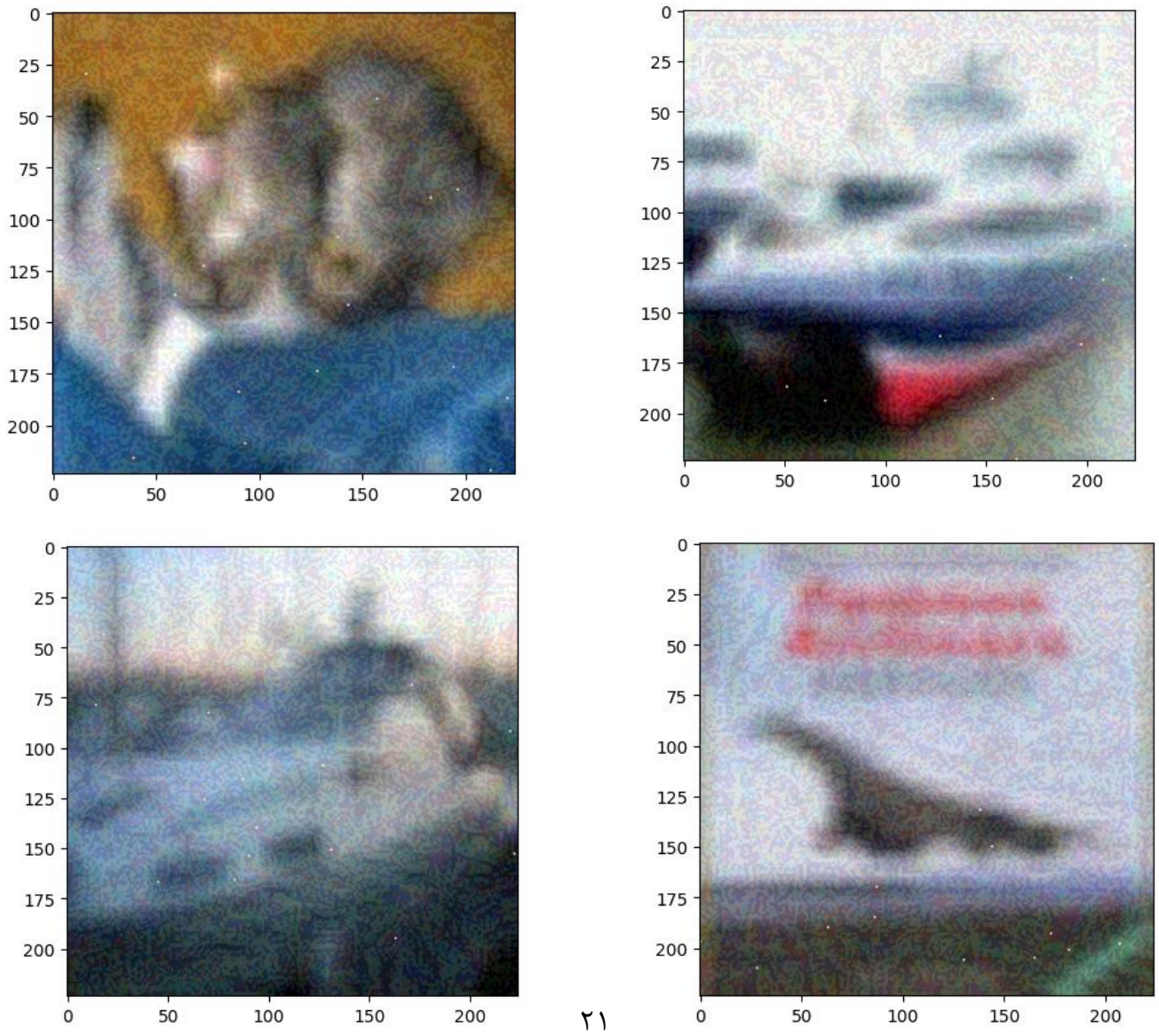
در رابطه بالا $\acute{x}_t$ ورودی مخرب در گام t، α اندازه گام در هر تکرار، Proj_{x+S} تابعی که ورودی ها را به داخل مجموعه قابل قبول S محدود می کند.

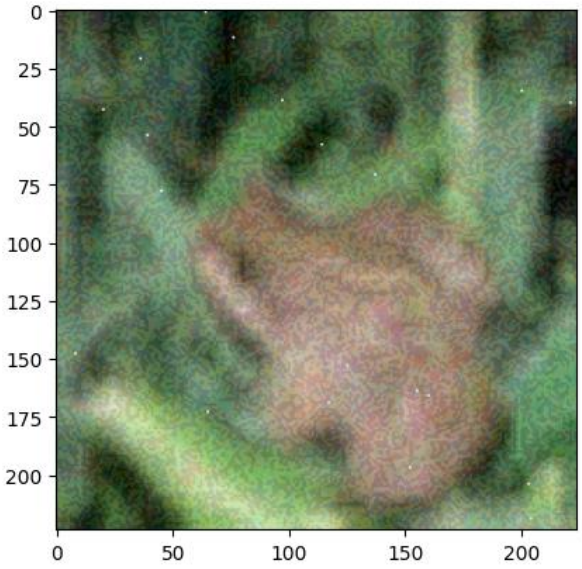
در ادامه میخواهیم تعدادی از تصاویر دادگان تست را به نمونه های adversarial تبدیل کنیم و نمایش دهیم.

برای این کار در ابتدا برای هر تصویر ۲۰ پیکس رندوم انتخاب می کنیم و رنگ آن ها را سفید می کنیم و در ادامه بر روی این تصاویر، الگوریتم FGSM (در کتابخانه cleverhans) را با استفاده از مدل Resnet18 از پیش آموزش دیده شده اعمال می کنیم.

```python
def add_noise(batch):
    output_tensors = []
    for image in batch:
        C, H, W = image.shape
        for _ in range(20):
            # Randomly choose a pixel
            h, w = np.random.randint(0, H), np.random.randint(0, W)
            # Change the pixel color to white
            image[:, h, w] = torch.tensor([1.0, 1.0, 1.0])
        output_tensors.append(image)
    return torch.stack(output_tensors, 0)
```

شکل ۵۰ – کد پیاده سازی اضافه کردن نویز رندوم

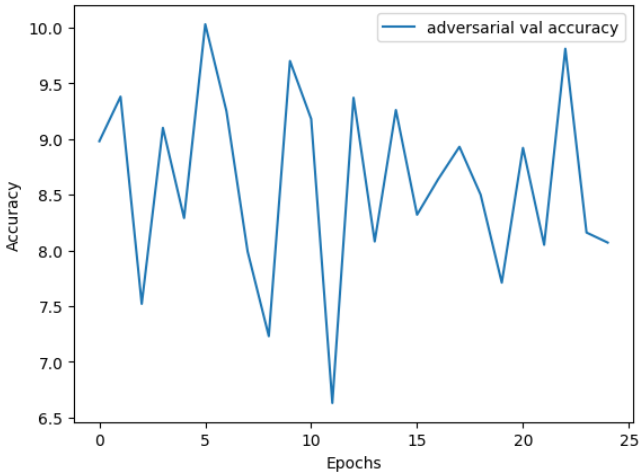




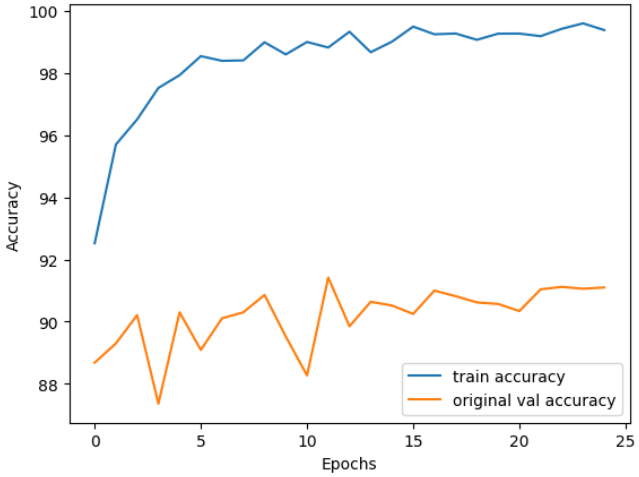
شکل ۵۱ – نمونه هایی از تصاویر adversarial

آموزش مدل با استفاده از داده Original و تابع هزینه Cross Entropy

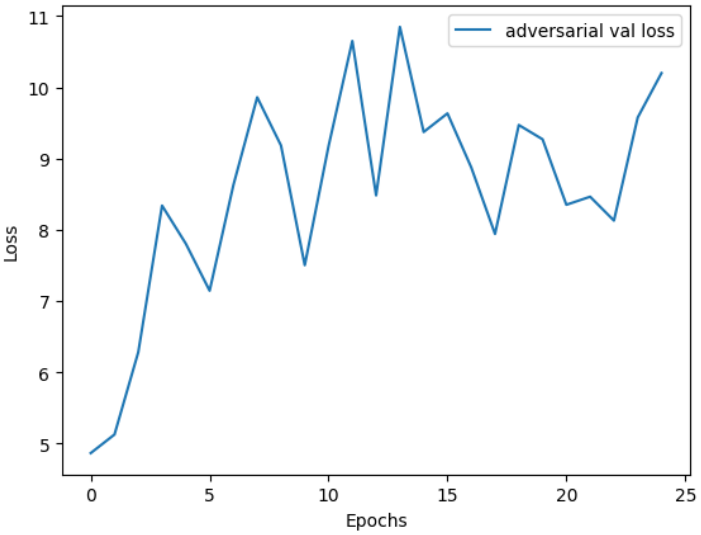
در این قسمت به تعداد ۲۵ ایپاک مدل از قبل آموزش دیده شده Resnet18 را روی داده های آموزش دیتاست CIFAR10 با تابع هزینه Cross Entropy آموزش می دهیم و آن را روی داده های تست تمیز و adversarial تست می کنیم.



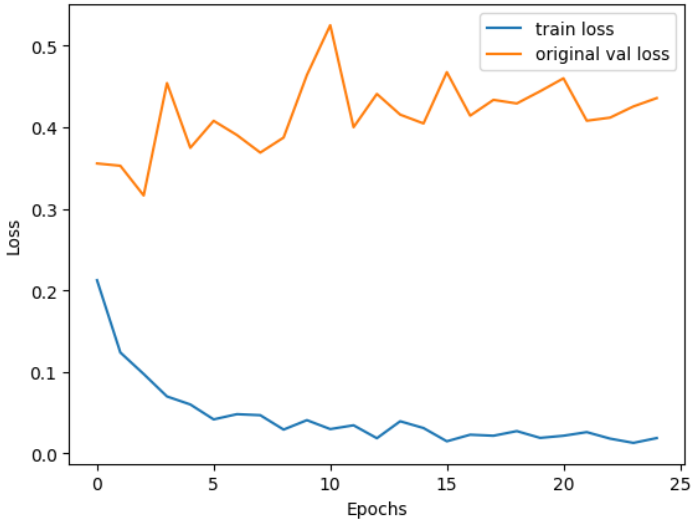
شکل ۵۲ – نمودار دقت داده ارزیابی adversarial



شکل ۵۳ – نمودار دقت داده آموزش و ارزیابی تمیز



شکل ۵۴ – نمودار خطای داده ارزیابی adversarial



شکل ۵۵ – نمودار خطای داده آموزش و ارزیابی تمیز

طبق تصاویر بالا می توانیم ببینیم که عملکرد مدل بر روی داده آموزش و داده ارزیابی تمیز طبیعی است (با وجود نوسان در داده ارزیابی تمیز) اما عملکرد آن بر روی داده ارزیابی adversarial به طور مشخص بد است و این یعنی روش FGSM کار خود را به درستی انجام داده است. همانطور که میبینیم دقت مدل بر روی داده ارزیابی adversarial تقریبا ۹ درصد و خطای آن در نهایت برابر با ۱۰ می باشد که عدد بزرگی است.

تست مدل بر روی داده تست تمیز:

در ابتدا دقت مدل آموزش دیده شده را بر روی داده تست تمیز محاسبه می کنیم:

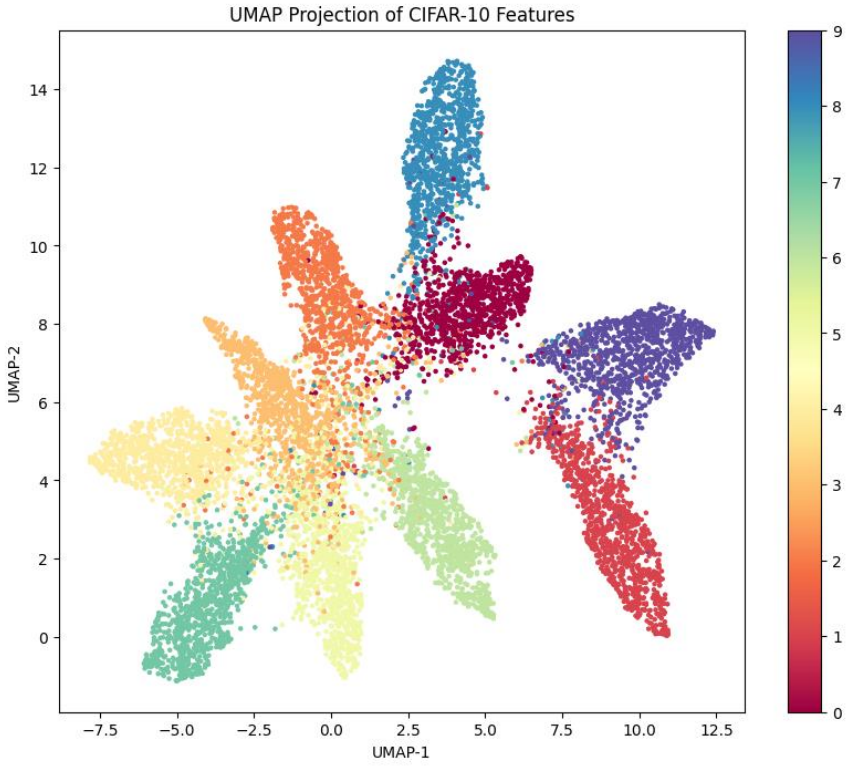
```
Accuracy of the model on the original test images: 89.31%
```

شکل ۵۶ – دقت مدل بر روی داده تست تمیز

در ادامه feature vector هایی که از طریق پاس دادن دیتای تمیز به مدل آموزش دیده شده بدست می آید (با بعد ۵۱۲) را از طریق روش UMAP به ۲ بعد کاهش می دهیم تا بتوانیم آن را نمایش دهیم:

```
reducer = umap.UMAP(n_neighbors=10, min_dist=0.3, n_components=2, random_state=42)
embedding = reducer.fit_transform(features.reshape(features.shape[0], -1))
```

شکل ۵۷ – کد پیاده سازی UMAP



شکل ۵۷ – نمایش دادگان توسط UMAP

همانطور که دقت مدل بر روی داده ی تست تمیز تقریبا ۹۰ درصد بود که دقت خوبی است می توان انتظار داشت که خروجی UMAP آن نیز خروجی قابل قبولی داشته باشد. طبق عکس شماره ۵۷ می توانیم ببینیم که تقریبا کلاس های مختلف به خوبی از هم جدا شده اند و یک کلاسترینگ قابل قبولی روی آن ها شکل گرفته است و به همین دلیل است که مدل می تواند به خوبی طبقه بندی روی این داده تست را انجام دهد.

تست مدل بر روی داده تست adversarial:

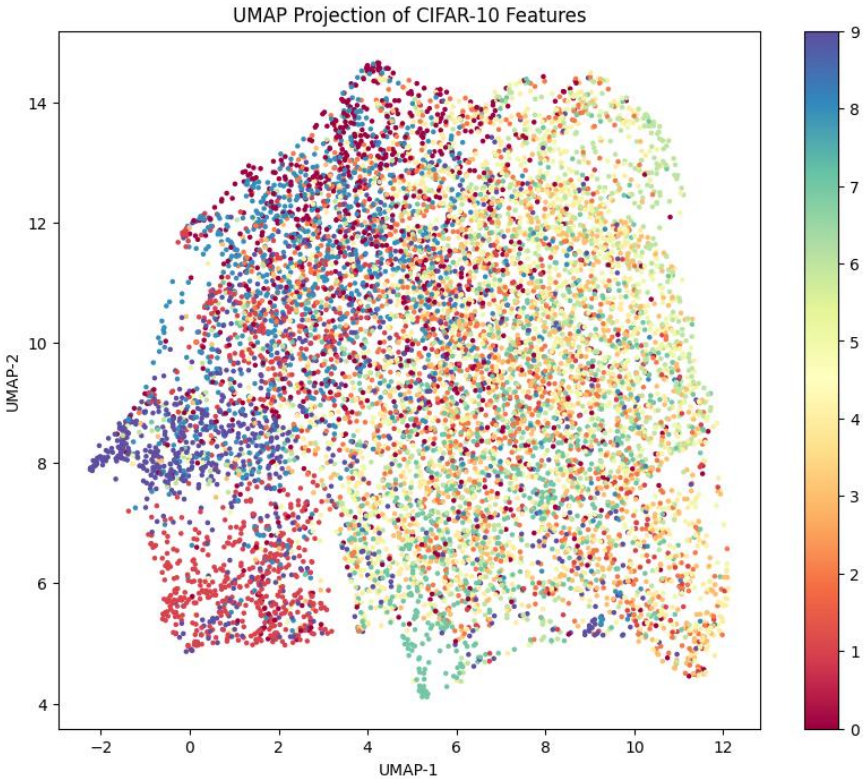
در ابتدا دقت مدل آموزش دیده شده را بر روی داده تست adversarial محاسبه می کنیم:

```
Accuracy of the model on the adversarial test images: 11.89%
```

شکل ۵۸ – دقت مدل روی داده تست adversarial

همانطور که انتظار می رفت مدل روی داده تست نویزی یا adversarial اصلا دقت خوبی ندارد و در واقع حمله FGSM به خوبی می تواند مدلی که موقع آموزش اصلا داده نویزی ندیده است را گول بزند.

در ادامه feature vector هایی که از طریق پاس دادن دیتای نویزی به مدل آموزش دیده شده بدست می آید (با بعد ۵۱۲) را از طریق روش UMAP به ۲ بعد کاهش می دهیم تا بتوانیم آن را نمایش دهیم:

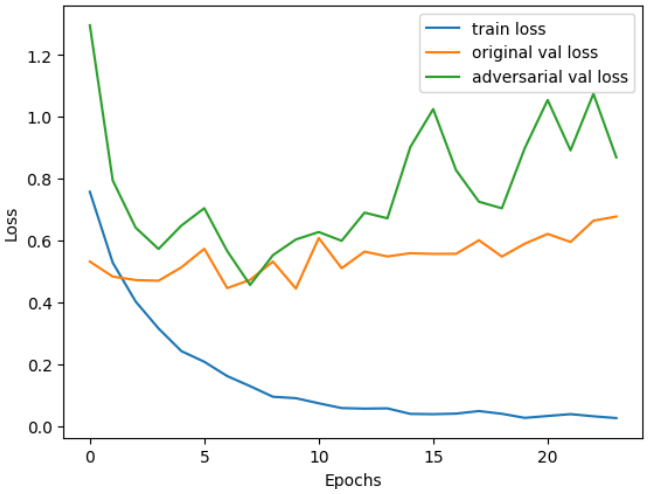


شکل ۵۹ – نمایش دادگان توسط UMAP

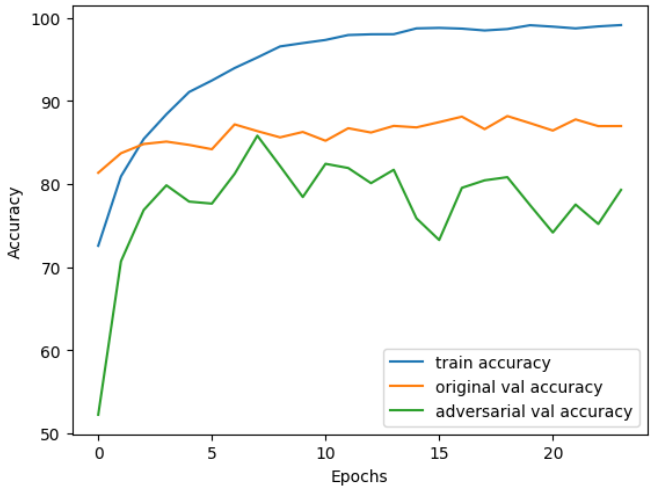
طبق عدد دقتی که در شکل شماره ۵۸ دیدیم انتظار نداریم تا خروجی UMAP دادگان خروجی مناسبی داشته باشد. تصویر شماره ۵۹ نشان می دهد که مدل اصلا نتوانسته است بین دادگان تست به خوبی تفاوت قائل شود و آن ها را طبقه بندی کند چون اکثر داده ها روی هم دیگر افتاده اند و قابل کلاسترینگ نیز نیستند.

آموزش مدل با استفاده از داده Augmented و تابع هزینه Cross Entropy

در این قسمت به تعداد ۲۵ ایپاک مدل از قبل آموزش دیده شده Resnet18 را روی داده های آموزش دیتاست CIFAR10 با تابع هزینه Cross Entropy آموزش می دهیم و آن را روی داده های تست تمیز و adversarial تست می کنیم. (در کد پیاده سازی این قسمت برای رسم نمودارهای دقت و خطا، لیست اعداد تشکیل دهنده ی این نمودارها به صورت دستی نوشته شده است. دلیل آن این است که آموزش مدل به دلیل طولانی بودن و چندین بار تست کردن و ران کردن و به محدودیت gpu کولب خوردن قطع می شود و runtime پاک می شد و اعداد از دست می رفت. بنابراین اعداد از طریق پرینت خروجی آموزش مدل وارد شده اند و نمودارها به درستی اطلاعات را نشان می دهند.)



شکل ۶۰ – نمودار خطای داده آموزش، ارزیابی تمیز و adversarial



شکل ۶۱ – نمودار دقت داده آموزش، ارزیابی تمیز و adversarial

همانطور که می بینیم برخلاف قسمت قبلی هر دو نمودار دقت و خطای هم برای داده ارزیابی تمیز و هم برای داده ارزیابی adversarial طبیعی است و فرایند آموزش عادی و درست را نشان می دهد که دلیل آن هم استفاده از داده ی adversarial در فرایند آموزش مدل است. طبق تصویر شماره ۶۱ دقت داده ارزیابی تمیز به تقریبا ۸۵ درصد و دقت داده ارزیابی adversarial به تقریبا ۸۰ درصد رسیده است.

تست مدل بر روی داده تست تمیز:

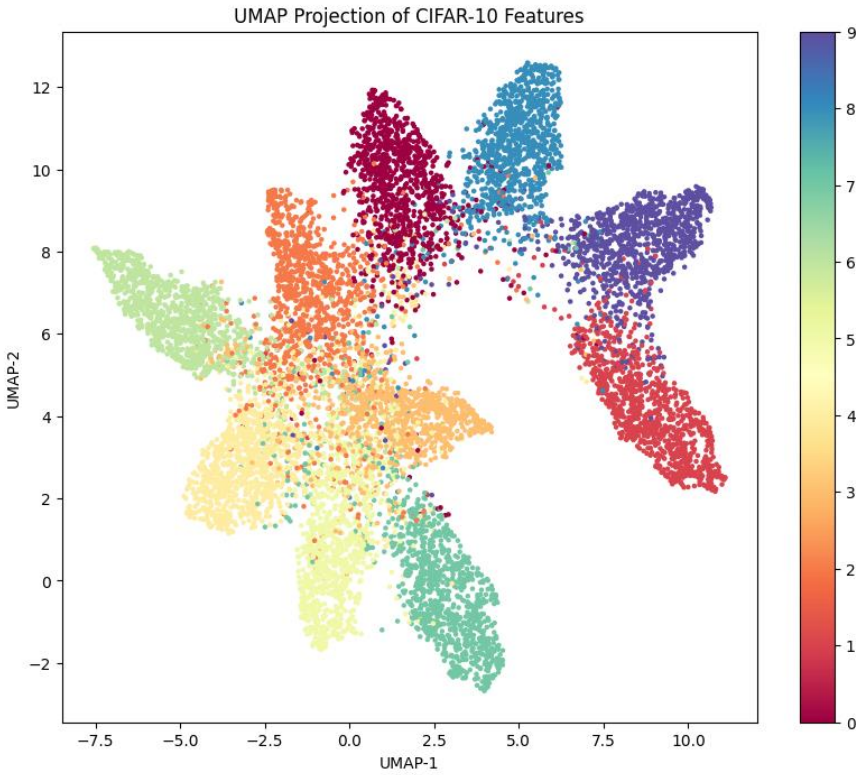
در ابتدا دقت مدل آموزش دیده شده را بر روی داده تست تمیز محاسبه می کنیم:

```
Accuracy of the model on the original test images: 87.33%
```

شکل ۶۲ – دقت مدل روی داده تست تمیز

طبق عکس بالا می توانیم ببینیم مدل آموزش دیده شده توسط adversarial training دقت خوبی روی داده تست تمیز دارد البته تقریبا ۲ درصد از حالت قبلی پایین تر است آن هم به این دلیل است که در ۵۰ درصد مواقع داده تمیز در فرایند آموزش استفاده نمی شود.

در ادامه feature vector هایی که از طریق پاس دادن دیتای تمیز به مدل آموزش دیده شده بدست می آید (با بعد ۵۱۲) را از طریق روش UMAP به ۲ بعد کاهش می دهیم تا بتوانیم آن را نمایش دهیم:



شکل ۶۳ – نمایش دادگان توسط UMAP

همانطور که دقت مدل بر روی داده ی تست تمیز تقریبا ۸۷ درصد بود که دقت خوبی است می توان انتظار داشت که خروجی UMAP آن نیز خروجی قابل قبولی داشته باشد. طبق عکس شماره ۶۳ می توانیم ببینیم که تقریبا کلاس های مختلف به خوبی از هم جدا شده اند و یک کلاسترینگ قابل قبولی روی آن ها شکل گرفته است و به همین دلیل است که مدل می تواند به خوبی طبقه بندی روی این داده تست را انجام دهد.

تست مدل بر روی داده تست adversarial:

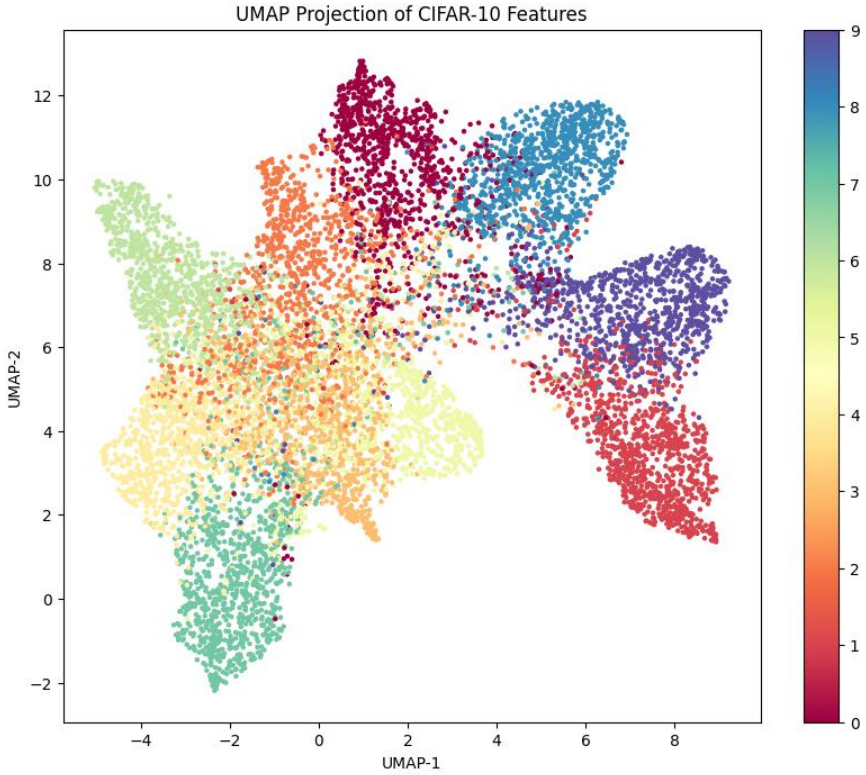
در ابتدا دقت مدل آموزش دیده شده را بر روی داده تست adversarial محاسبه می کنیم:

```
Accuracy of the model on the adversarial test images: 79.65%
```

شکل ۶۴ – دقت مدل روی داده تست adversarial

همانطور که می بینیم دقت مدل روی داده تست نویزی نسبت به حالت قبل به شدت افزایش داشته است و به تقریبا ۸۰ درصد رسیده است. بنابراین می توان گفت روش adversarial training کار خودش را به درستی انجام داده است و مدل نسبت به داده ی نویزی این دیتاست تقریبا مقاوم یا Robust شده است.

در ادامه feature vector هایی که از طریق پاس دادن دیتای نویزی به مدل آموزش دیده شده بدست می آید (با بعد ۵۱۲) را از طریق روش UMAP به ۲ بعد کاهش می دهیم تا بتوانیم آن را نمایش دهیم:



شکل ۶۵ – نمایش دادگان توسط UMAP

همانطور که دقت مدل بر روی داده ی تست نویزی تقریبا ۸۰ درصد بود که دقت خوبی است این دفعه بر خلاف دفعه قبل می توان انتظار داشت که خروجی UMAP آن نیز خروجی قابل قبولی داشته باشد. طبق عکس شماره ۶۵ می توانیم ببینیم که تقریبا کلاس های مختلف به خوبی از هم جدا شده اند و یک کلاسترینگ قابل قبولی روی آن ها شکل گرفته است و به همین دلیل است که مدل می تواند به خوبی طبقه بندی روی این داده تست را انجام دهد.

تابع هزینه Circle Loss

تابع هزینه Circle Loss به طور خاص برای بهبود مدل‌هایی که وظیفه‌ی تشخیص و تمایز بین دسته‌های مختلف داده‌ها را دارند طراحی شده است. این تابع هزینه به‌گونه‌ای طراحی شده که امکان یادگیری فضای ویژگی‌ها را بهینه‌سازی کند تا نمونه‌های متعلق به یک کلاس نزدیک به هم و نمونه‌های متعلق به کلاس‌های مختلف دور از هم قرار گیرند. این کار با استفاده از تأکید بر فاصله‌های زاویه‌ای انجام می‌شود. در ادامه به چند فایده استفاده از این تابع هزینه اشاره می کنیم:

بهبود تمایز بین کلاس‌ها: تابع هزینه Circle Loss به گونه‌ای طراحی شده که تشخیص و تفکیک بین کلاس‌های مختلف را بهینه‌سازی کند، که منجر به بهبود دقت مدل در تشخیص دسته‌های مختلف می‌شود.

کاهش حساسیت به نمونه‌های پرت: با تمرکز بر روی فواصل زاویه‌ای، این تابع هزینه تأثیر نمونه‌های پرت را در یادگیری کاهش می‌دهد.

روش‌های قدیمی‌تر مانند Cross-Entropy Loss بر مبنای تفاوت بین برچسب واقعی و برچسب پیش‌بینی‌شده کار می‌کنند، که ممکن است در مواردی که تمایز بین کلاس‌ها به‌ویژه مهم است، کافی نباشد. تابع هزینه Circle Loss با ارائه تاکید بیشتر بر روی جدایی زاویه‌ای و تمایز بین کلاس‌ها، این محدودیت‌ها را برطرف می‌کند. Circle Loss با تمرکز بر روی کمینه‌سازی فاصله‌های زاویه‌ای بین نمونه‌های مختلف، به بهبود استحکام ویژگی‌های استخراج شده کمک می‌کند. این امر باعث می‌شود که تغییرات جزئی در ورودی، که معمولاً توسط حملات adversarial ایجاد می‌شوند، تأثیر کمتری بر روی خروجی نهایی داشته باشند.

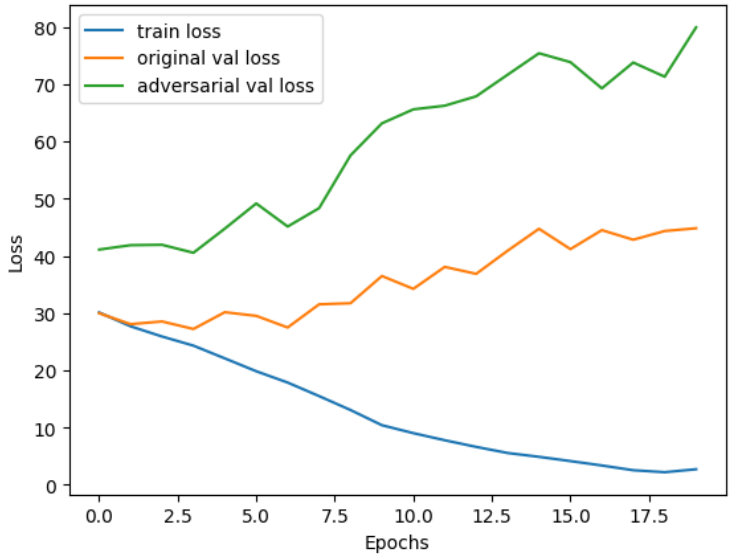
آموزش مدل با استفاده از داده Original و تابع هزینه Circle Loss

در ابتدا چون تابع هزینه را به Circle Loss تغییر داده ایم نیاز است تا لایه آخر مدل Resnet18 (Fully Connected) را حذف کنیم تا بتوانیم خروجی AveragePooling تصاویر هر batch را داشته باشیم و آن را به همراه لیبل های مرتبط به Circle Loss بدهیم. همچنین برای پیاده سازی Circle Loss از کتابخانه Pytorch_metric_learning.losses استفاده می کنیم و طبق مقاله آورده شده، پارامترهای آن را به صورت پارامترهای دیفالت m = 0.4 و gamma = 80 قرار می دهیم.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = models.resnet18(pretrained=True)
num_classes = 10
model.fc = torch.nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)
model = torch.nn.Sequential(*(list(model.children())[:-1]))
```

شکل ۶۶ – حذف لایه آخر شبکه Resnet18

در ادامه به تعداد ۲۰ ایپاک مدل از قبل آموزش دیده شده Resnet18 (با حذف لایه آخر) را روی داده های آموزش دیتاست CIFAR10 با تابع هزینه Circle Loss آموزش می دهیم و آن را روی داده های تست تمیز و adversarial تست می کنیم.



شکل ۶۷ – نمودار خطای داده آموزش، ارزیابی تمیز و adversarial

در این بخش برای بدست آوردن دقت نمی توانیم مانند بخش های قبل عمل کنیم زیرا مدل ما خروجی یک بردار ۵۱۲ تایی (خروجی AveragePooling) می دهد نه بردار ۱۰ تایی. بنابراین نیاز است تا از روش KNN برای بدست آوردن دقت استفاده کنیم.

در ابتدا با استفاده از پاس دادن دیتای آموزش به مدل آموزش دیده شده، Feature Vector های تصاویر آموزش را بدست می آوریم و ذخیره می کنیم. سپس با استفاده از این بردار های و لیبل های مرتبط با آن ها یک مدل KNN را فیت می کنیم.

```
### Get embeddings of train images ###
model.eval()
embeddings = []
labels = []
with torch.no_grad():
  for images, targets in train_loader:
    images = images.to(device)
    targets = targets.to(device)
    features = torch.squeeze(model(images))
    embeddings.append(features)
    labels.append(targets)
train_embeddings = torch.cat(embeddings)
train_labels = torch.cat(labels)
train_embeddings = train_embeddings.view(train_embeddings.size(0), -1)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(train_embeddings.cpu().numpy(), train_labels.cpu().numpy())
```

شکل ۶۸ – بدست آوردن بردارهای ویژگی تصاویر آموزش و ساخت مدل KNN

در ادامه برای بدست آوردن دقت این مدل بر روی داده های تست، ابتدا داده های تست را به مدل می دهیم تا بردارهای ویژگی آن تصاویر را استخراج کنیم و سپس با استفاده از متد Predict در مدل KNN آموزش دیده، لیبل های پیش بینی شده را بدست می آوریم و با لیبل های اصلی مقایسه می کنیم و در نهایت دقت را می توانیم بدست آوریم.

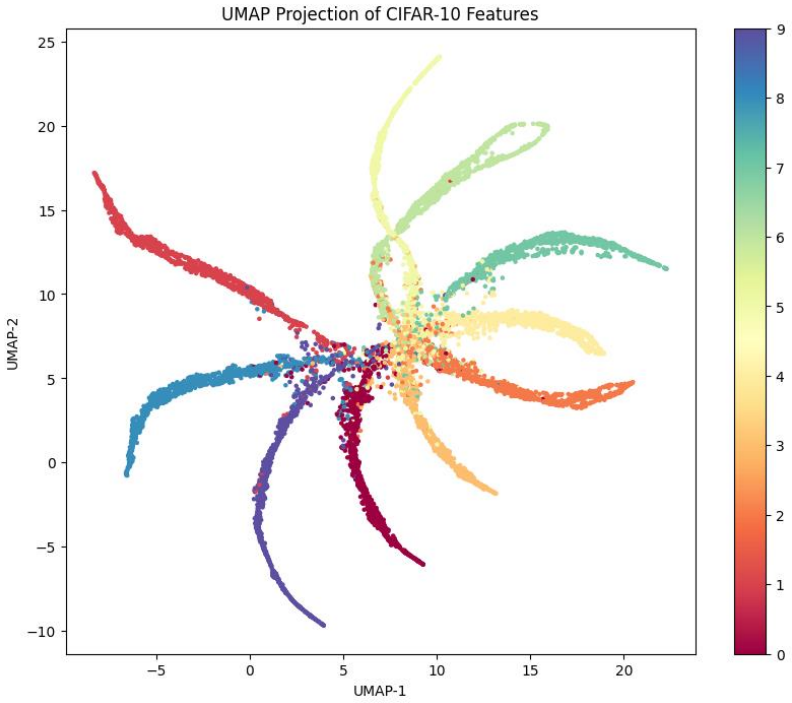
تست مدل بر روی داده تست تمیز:

```
Test Accuracy on original test images: 0.8952
```

شکل ۶۹ – دقت مدل روی داده تست تمیز

طبق تصویر بالا می توانیم ببینیم که با استفاده از تابع خطای Circle Loss توانسته ایم بهترین دقت را برای داده تست تمیز تا اینجای کار بگیریم.

در ادامه feature vector هایی که از طریق پاس دادن دیتای تمیز به مدل آموزش دیده شده بدست می آید (با بعد ۵۱۲) را از طریق روش UMAP به ۲ بعد کاهش می دهیم تا بتوانیم آن را نمایش دهیم:



شکل ۷۰ – نمایش دادگان توسط UMAP

طبق عکس بالا می توانیم ببینیم تابع خطای Circle Loss در اینجا نیز بهتر از تابع خطای Cross Entropy عمل کرده است زیرا بهتر توانسته است کلاس ها را از یکدیگر جدا کند و کلاسترینگ بهتری ارائه دهد. دلیل آن هم این است که این تابع خطا بر روی زیاد کردن فاصله بین کلاس ها و کمینه کردن فاصله درون کلاسی تمرکز می کند.

تست مدل بر روی داده تست adversarial:

```
Test Accuracy on adversarial test images: 0.0728
```

شکل ۷۱ – دقت مدل روی دیتای تست adversarial

طبق نتایجی که تا اینجای کار دیدیم، بهترین دقت و طبقه بندی داده های تمیز متعلق به وقتی بود که داده های آموزش را با تابع خطای Circle loss آموزش دادیم و توانست جداسازی دقیق تری روی داده ها انجام دهد. اما اگر بخواهیم مدل Robust تری نسبت به حمله ها از جمله FGSM داشته باشیم بهترین

روش adversarial training بود که دیتای نویزی در آموزش مدل استفاده شدند و به دقت تقریبا ۸۰ درصد روی داده ی تست نویزی رسیدیم.

مراجع

1. https://cafetadris.com/blog/%D9%BE%DB%8C%D8%B4%DA%AF%DB%8C%D8%B1%DB%8C-%D8%A7%D8%B2-%D8%A8%DB%8C%D8%B4%D8%A8%D8%B1%D8%A7%D8%B2%D8%B4-%D8%AF%D8%B1-%DB%8C%D8%A7%D8%AF%DA%AF%DB%8C%D8%B1%DB%8C-%D8%B9%D9%85%DB%8C%D9%82/
2. https://deeplearning.ir/%D9%85%D8%B9%D8%B1%D9%81%DB%8C-batchnormalization/
3. https://blog.faradars.org/adversarial-attacks-fgsm/
4. N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014
5. S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” proceedings.mlr.press, Jun. 01, 2015. https://proceedings.mlr.press/v37/ioffe15.html
6. Yifan Sun, Changmao Cheng, Yuhan Zhang, Chi Zhang, Liang Zheng, Zhongdao Wang, and Yichen Wei. Circle loss: A unified perspective of pair similarity optimization. In CVPR, pages 6398–6407, 2020.